Please cite as: UC Berkeley ERL MEMO M03/22

Implementation Issues in Hybrid Embedded Systems

Stephen Neuendorffer

EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.

June 24, 2003

Abstract

This paper presents an approach to the implementation of electronic computation systems whose behavior is tightly integrated with the physical world. We call such systems *hybrid embedded systems*. Such systems are challenging from a design perspective because their behavior is governed by both continuous-state dynamics from the physical world and discrete-state dynamics from the computation. There are several difficulties that appear in such systems. For instance, understanding of the passage of time during computation is critical to understanding how the computation system affects the state of the physical world. Hybrid embedded systems are also inherently concurrent; the computation system operates concurrently with the dynamics of the physical world, in addition to any concurrency that may be designed into the system. In addition, hybrid embedded systems must generally operate within the constraints of traditional embedded systems. They are inevitably constrained computationally, often have a complex computational architecture, and must perform predictably. This paper presents an approach to the design of embedded systems utilizing component-based system models capable of representing concurrency, the passage of time, and both continuous and discrete behaviors. These models allow for automatic generation of system implementations from high-level abstractions as well as the consideration of low-level architectural details where necessary. We

show how this technique can be ued to approach difficulties in the design of a complex digital control system.

1 Introduction

Hybrid system formalisms are often used to simply to represent and describe the behavior of a real world physical system that cannot be easily described as a differential system. This includes many of the commonly cited hybrid systems examples, such as the two water tanks example, or the bouncing ball example. Less commonly, hybrid system formalisms are used to describe the interaction between discrete computation systems and the physical world. This interaction often arises in the form of *embedded systems*, such as digital control systems, [10, 4] high-performance data acquisition systems, [11] and heterogeneous electronic systems containing analog and digital components. [9] In these cases, the interaction between the discrete and analog portions of a system are tightly coupled and crucial to the proper behavior of the system. We would like to model the behavior of such systems abstractly as hybrid systems in order to quickly design the function of discrete computation. In addition, a hybrid system formalism can help a designer to understand and perhaps verify the overall behavior of the resulting system. We call such a model a *hybrid embedded system*.

Ptolemy II is a software modeling tool that is particularly effective for modeling hybrid embedded systems. [2] This effectiveness arises primarily from an emphasis on *hierarchical heterogenity*. [3] Hierarchical heterogeneity allows discrete-state and continuous-state models to be constructed from primitive components according to high-level patterns of component interaction, called *models of computation*.[8] Additionally, these models can themselves be viewed as components with an opaque interface. Hence, large models can be constructed from smaller models. Since control flow and data flow between components in a model is restricted by the component interface, detailed system models can be constructed without overwhelming a designer. Depending on the model of computation, models can represent systems with either continuous or discrete state, making them useful for representing hybrid embedded systems. Additionally, concurrent models of computation can represent computational structures which are difficult to represent and reason about using current hybrid systems techniques, such as Hybrid I/O automata.

We call this style of component-based modeling *actor-oriented design*. One advantage of actor-oriented modeling is abstraction: systems can be represented

at high levels of abstraction, facilitating rapid creating of executable simulation models. However, more detailed models can also be constructed containing details necessary for implementation synthesis. If we are modeling a software system, then we might say that *the model is a program*. Implementation synthesis, or *automatic code generation*, is not fundamentally different from the compilation process that is familiar to software engineers. The fundamental difference is that actor-oriented modeling emphasizes the interaction between different parts of a system, rather than the behavior of individual portions. Timing and concurrency can be represented directly as part of an actor-oriented model, rather than being considered as secondary aspects in an untimed, sequential language.

Our approach to solving this problem involves several aspects. We avoid error-prone implementation through *automatic code generation* that starts from abstract models. Furthermore, we require designers to model, at a relatively detailed level, implementation-related aspects of a system that are often abstracted away in hybrid system models. Since low-level modeling can be time consuming, we allow models to interoperate at different levels of detail, to allow refinement from abstract models to more detailed models. Lastly, we concentrate on providing modeling abstractions that are appropriate for hybrid embedded systems, with semantics that represent software delays and timing requirements.

This paper shows how actor-oriented design techniques, as embodied in [2] can be used to approach the design and implementation of a hybrid embedded system. The design techniques will be illustrated using the Caltech Multi-vehicle Wireless Testbed [1] as an extended example. By using different models of computation, models will represent not only the continuous dynamics of these vehicles but also the concurrent interaction of a distributed control system. We will present models for system simulation, models for automatic code generation, and also show how a designer can make use of intermediate refinements for hardware-in-the-loop simulation.

2 Caltech Multi-Vehicle Wireless Testbed

Murray *et al.* at Caltech have developed a platform for experimenting with coordinated control of autonomous vehicles, called the MVWT. [1] The platform consists of a number of ground vehicles operating in a controlled environment. Propulsion for each vehicle is provided by a pair of ducted fans mounted on top of the vehicle. By applying the same force to each fan, the vehicle will move forward in a straight line, while applying a different force to each fan causes the vehicle to

turn. An embedded computer controls the fans through off-the-shelf motor controllers connected to an RS-232 interface. The vehicles slide on three industrial casters, allowing them to slide sideways while turning. The operating environment of the vehicles includes a video camera-based localization system (the *Lab Positioning System*), which broadcasts location and orientation information for each vehicle over 802.11b wireless ethernet using UDP datagrams.

In many ways the Caltech system is an appealing environment for experimenting with embedded digital control. Primarily, the continuous dynamics of the system are qualitatively different from the dynamics of a 'steered vehicle' such as a car, which cannot (generally) slide sideways when turning. The vehicle dynamics are more similar to a two dimensional approximation of winged aircraft dynamics. Additionally, since the vehicles are small and highly mobile, it is difficult to interact directly with executing control code. Control commands, such as way points, are entered from a separate base station computer and transmitted to the control algorithm running on a vehicle. Perhaps most importantly, the MVWT is both safe and cheap to operate, offering a distinct advantage over flying vehicles. [5]

From a control-oriented viewpoint, the continuous-time dynamics of a Caltech vehicle can be modeled by the equations (from [1]):

$$m\ddot{x} = -\eta\dot{x} + (F_s + F_p)\cos\theta$$

$$m\ddot{y} = -\eta\dot{y} + (F_s + F_p)\sin\theta$$

$$J\ddot{\theta} = -\psi\dot{\theta} + (F_s - F_p)r$$

where friction is assumed to be proportional to velocity. F_s and F_p are inputs to the system corresponding to the forces applied to the starboard and port fans respectively. The system state $\sigma = [x, \dot{x}, y, \dot{y}, \theta, \dot{\theta}]^T$, is available to a control algorithm through the localization system. The Caltech group has implemented LQR-based state-feedback digital control of the above dynamics capable of tracking trajectories.

In the physical system, there are several hardware limitations that serve to complicate the design of the digital control system. For instance, in the physical system, F_s and F_p are limited to a maximum of approximately 5 Newtons, and cannot operate in reverse. Additionally, the fans are driven by discrete-input motor controllers, resulting in quantization of the forces that can actually be applied to the vehicle. Further constraints are caused by the fact that the control software and localization system cannot operate continuously. The localization system is

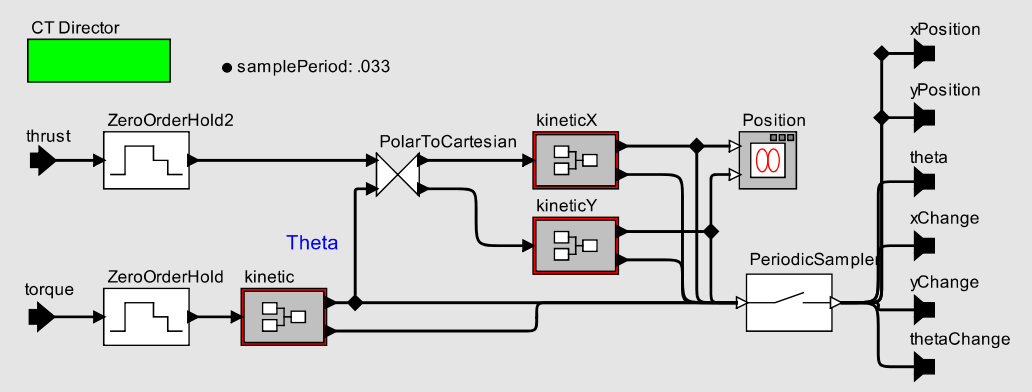

Figure 1: A model of the continuous dynamics of a single vehicle.

only capable of capturing 60 frames of video per second, limiting the availability of location estimates. Lastly, the system incorporates concurrent, distributed computation for computing the control law. Furthermore, this distributed computation operates over a communication channel that has the potential to lose data.

3 Basic Control Model

A translation of the dynamics of a single vehicle into a Ptolemy II model is shown in Figure 1. This model is constructed in the style of Simulink, a commercial tool produced by The Mathworks. The semantics of component interaction are designed to support numerical integration algorithms, implemented by the `Integrator` component. The signals communicated between components are interpreted as functions of time that are solutions to a set of Ordinary Differential Equations. We call these signals *continuous-time signals*.

In this model, the inputs are not taken as continuous functions, but are instead assumed to be discrete-event signals. *Discrete-event signals*, unlike continuous-time signals, are assumed to take values only at a countable number of points in time. At all other points in time, a discrete-event signal has no value and is said to be *absent*. The `Zero-Order Hold` components convert from discrete-event signals into continuous-time signals suitable for integration. Similarly, the `PeriodicSampler` component produces a discrete-event signal, which happens to consist of events evenly spaced in time, from the continuous-time signal.

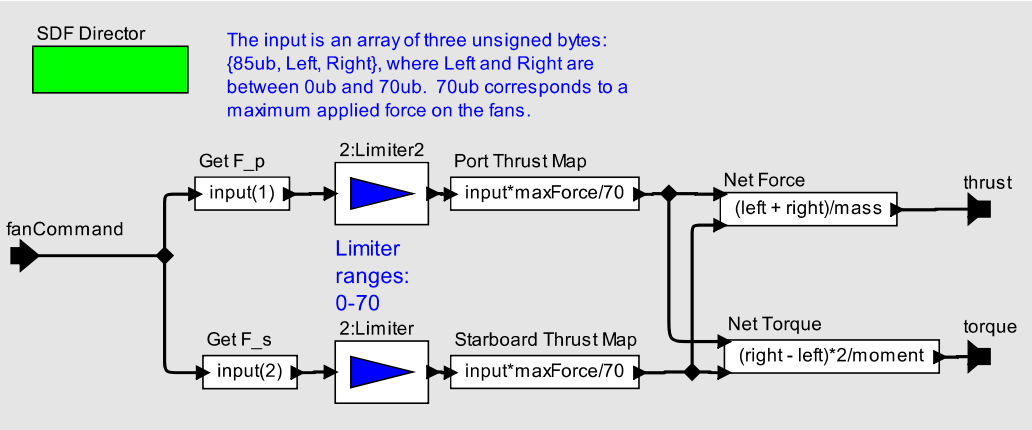

Figure 2: Computation of the thrust and torque, given an array of three unsigned bytes from the controller.

In this model, the `PeriodicSampler` models the fact that only sample values of the continuous-time dynamics are available to the vehicle controller.

The `thrust` and `torque` inputs are computed by the model in figure 2. The input to this model is a discrete-event signal, where the value of each event is an array of three unsigned bytes, corresponding to a control signal received by the vehicle from the control computer over RS-232. The first byte is a fixed start byte, and is ignored. The next two bytes correspond to an applied force for the port and starboard fans. This model is a stateless model, and computes a simple function of the three input bytes. A similar model (not shown) takes the output of the `PeriodicSampler` and constructs another unsigned byte array encoding the six state variables (in 64-bit IEEE floating point format) and the current time. This array corresponds exactly in structure to the array of bytes constructed by Caltech's Lab Positioning System.

The interaction between the vehicle model above, and the controller is shown in Figure 3. This model includes a detailed model of the data format between the plant and the controller. The localization system broadcasts a UDP datagram containing the encoded state of the vehicle, approximately 60 times a second. This communication is modeled by the discrete-event signal connection between the output of the vehicle model and the input of the controller model. In response to each network packet, the control algorithm performs come computation and eventually sends a three byte serial sequence to change the speed of the fans. The

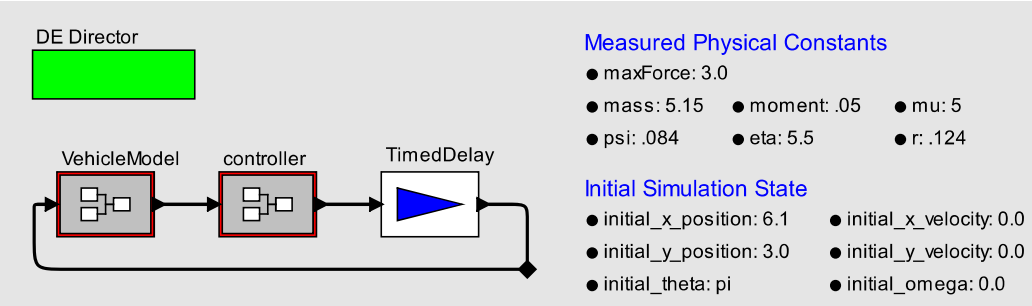

Figure 3: The toplevel simulation model, showing the interaction between the model of vehicle dynamics and the model of the controller.

serial communication is modeled by a separate event sent out from the controller. Unfortunately, from the point of view of accurate simulation, we have no idea how long this computation will actually take in the final system. In this model, the controller is idealized and generates its output event in zero time. The model includes an explicit model of the computation and communication delay, given by the `TimedDelay` component. Here the delay is assumed to be constant, but it is trivial to substitute a stochastic delay, perhaps allowing for the possibility of dropped packets.

The model of the controller itself is shown in Figure 4. This controller is based on an LQR controller design by Murray's group for tracking circular trajectories. The trajectories are generated in polar form according to parameters specified in the model. Note that input to the `Circular Trajectory Controller` is a structured record datatype containing six fields, one for each state variable of the physical system. The state is converted to another record containing the state in polar form and the control law is computed in polar space. A simulation plot is shown in Figure 5.

4 Implementation

The model in previous section is a model that is constructed primarily to allow simulation. Some aspects of the intended system have been modeled explicitly, such as the format of information received from the localization system. On the other hand, some aspects of the system have been abstracted, such at the communication between the localization system and the controller. The model represents

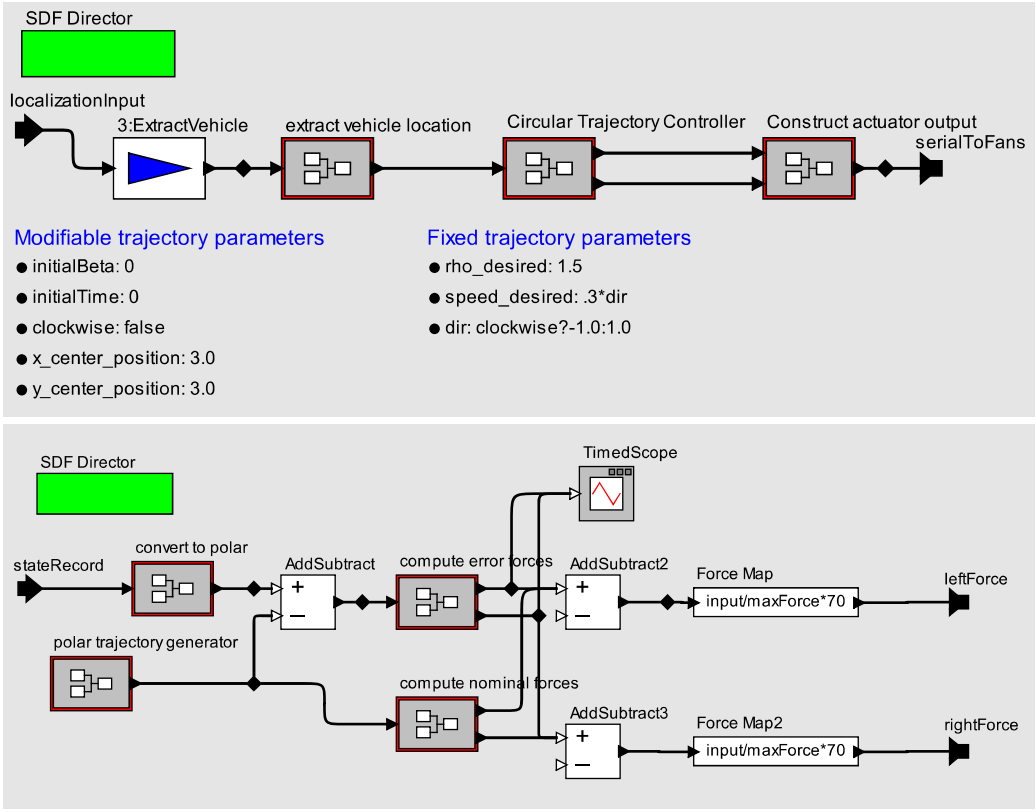

Figure 4: Model of the vehicle controller. Most of the components decode localization information from an array of bytes into a record of values, and encode the control output back into an array of bytes. The interesting part of the controller is implemented by the `Circular Trajectory Controller`, and is shown expanded below.

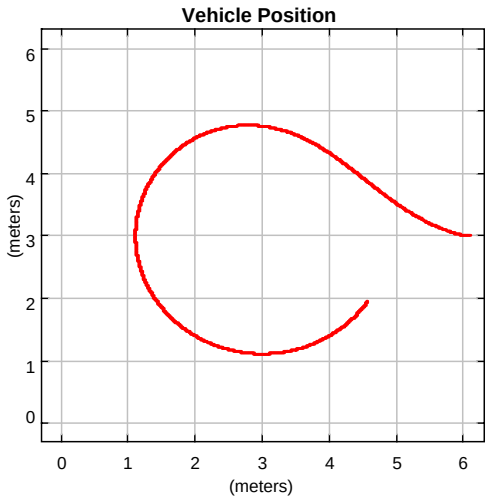

Figure 5: A simulation plot of the position of a vehicle, tracking a counter-clockwise circular trajectory around the point (3,3).

this communication as an instantaneous event, while the actual communication layer incurs some random (possibly infinite) delay. The model of the vehicle is, itself an abstract representation of the vehicle and the localization system. These models cannot be viewed as a program, i.e. a source for synthesis, without additional information, such as a communication protocol or a 3D CAD model of the physical vehicle.

The controller on the other hand, is a concrete model. Given appropriate inputs and outputs, the controller is in a form which directly corresponds to a software architecture for implementing the controller algorithm. This architecture can be automatically generated in a relatively straightforward mapping from the original. However, in order to perform the synthesis procedure, it is necessary to separate the concrete, synthesizeable portion of the simulation model from the abstract portion. This process is known as *system partitioning*. The result of partitioning the above system is shown in Figure 6. Note that the communication channels have been replaced with `Datagram`, and `SerialComm` components encapsulating the UDP and RS-232 communication interfaces.

Note that Figure 6 includes the entire partitioned model, including the abstract portion corresponding to the vehicle dynamics. While this portion is not useful for implementation synthesis, it can be used for distributed simulation. By executing the model of the vehicle on one computer and a model of the controller on another

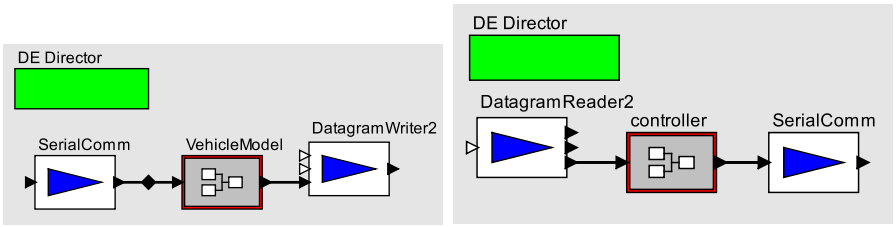

Figure 6: A partitioned version of the simulation model, where event communication has been replaced with communication interfaces. The `VehicleModel` and `Controller` models are as before.

computer, more accurate simulation of the system behavior can be performed. In particular, such a simulation includes actual properties of the communication protocol, rather than the approximate model included earlier. Other combinations are also possible, resulting in various forms of hardware-in-the-loop simulation. For instance, the controller model can be executed in the actual system, taking the place of an embedded controller. This structure allows us to test that communication protocols and vehicle dynamics have been modeled in sufficient detail. Alternatively, the vehicle dynamics model can be executed with code generated from the controller model, to test that the implementation was generated correctly.

5 Improving the System Model

The model presented above includes many details that are abstracted by the differential equation dynamics. However, from an embedded software perspective, the model is still very minimal. It does not model how the system is initialized, for instance, or how the system recovers from errors. This information must either be specified as part of code generation, perhaps by specifying a target platform that provides initialization and reset capabilities, or it must be specified through a more detailed system model. In order to show how these might be represented in a more detailed model, we concentrate on the interaction between the control algorithm and the base station computer.

The first interaction that we attack is the ability to trigger mode switches from the base station. This is modeled by augmenting the model of the controller to include a multi-modal controller, as shown in Figure 7. In each mode, the modal controller behaves as the original controller, which follows circular trajec-

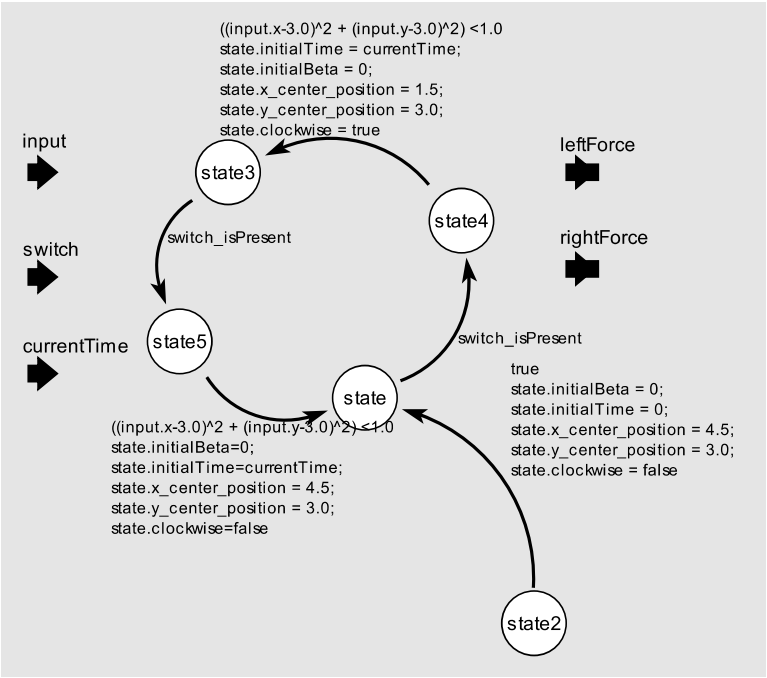

Figure 7: A model of a modal controller. The base station computer can send an event over the network, causing this controller to make a state transition, updating the parameters of the trajectory being followed.

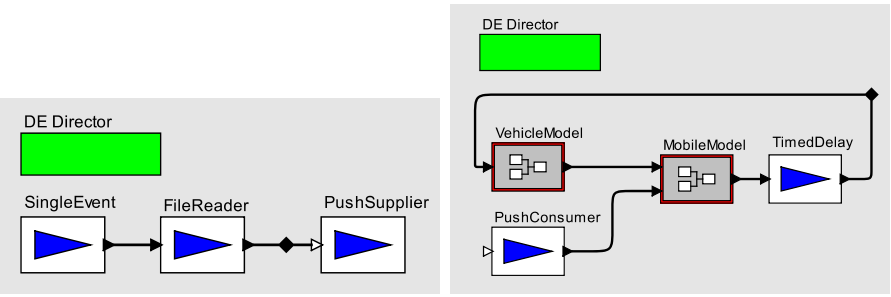

Figure 8: A model of the interaction between the base station computer (on the left) and the vehicle on the right. The base station can remotely update the controller being executed.

tory. However, when the modal controller switches modes, the parameters of the trajectory can be modified, allowing the vehicle to follow two different circular trajectories. The mode switch is triggered by an event, received from the base station over the wireless network. However, the new trajectory is not started until the position of the vehicle is close enough to the new trajectory to allow a safe transition. This is represented by having an intermediate state between the reception of the trigger event, and resetting the trajectory parameters of the controller.

The second interaction that we deal with is the ability of the base station to dynamically update and modify the control algorithm remotely. This is modeled using a `MobileModel`, as shown in Figure 8. This component does not have behavior of its own, but simply encapsulates other components received on its second input port. In this case, the mobile model receives a description of the component over a CORBA-based publish and subscribe network, represented by `PushConsumer` and `PushSupplier`. Essentially, the controller publishes a event service which the base station computer subscribes to, allowing it to push a new component description to the controller. Currently the component description is an XML fragment, but we are working allowing serialized Java components with code to be transmitted over the network. Additionally it seem straightforward to combine controller updates with modal models to ensure that switching to an updated controller occurs safely.

6 Related Work

The work presented here integrates several areas of research. Physical simulation tools such as Simulink and Modelica [13] allow designers to represent physical dynamics. The Simulink Real-time workshop allows code to be generated from a discrete-state controller model, but primarily targets single processor, RTOS-based systems. Some effort has been made to create better programming abstractions for real-time systems, such as the Giotto language, [6] and graphical representations of Giotto [7]. One advantage of Giotto is that it can be compiled to run in a distributed computing environment. Hardware-in-the-loop simulation is commonly used in industry, in order to test hardware prior to system deployment. However these systems can be difficult to develop and maintain, hence a need for hardware-in-the-loop frameworks that can be reused from one project to the next. [12]

7 Conclusion

This paper presents a technique for designing and implementing hybrid embedded systems, based on the composition of components. We approach the difficulty that the design of a control system is incompletely dealt with through analytical techniques, and by existing simulation frameworks. Our approach is based on system simulation and refinement into a system model that can be compiled into embedded software. Non-ideal aspects of a system can be modeled and integrated with ideal models for system simulation. Hardware-in-the-loop simulation can reduce dependence on idealized models and improve our understanding of the resulting system before synthesizing an implementation. We have shown how these ideas can be incorporated into a generic framework for component composition and code generation.

References

[1] L. Cremean et al. The Caltech multi-vehicle wireless testbed. In *Proceedings of the Conference on Decision and Control (CDC)*. IEEE, Dec. 2002.

[2] J. Davis et al. Ptolemy II - Heterogeneous concurrent modeling and design in Java. Memo M01/12, UCB/ERL, EECS UC Berkeley, CA 94720, Mar. 2001.

[3] J. Eker et al. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1), Jan. 2003.

[4] J. Eker, C. Fong, J. W. Janneck, and J. Liu. Design and simulation of heterogeneous control systems using Ptolemy II. In *IFAC Conference on New Technologies for Computer Control*, Hong-Kong, China, November 2001. IFAC.

[5] J. Evans, G. Inalhan, J. Jang, R. Teo, and C. Tomlin. Dragonfly: A versatile UAV platform for the advancement of aircraft navigation and control. In *Proc. of the 20th Digital Avionics Systems Conference*. IEEE, Oct. 2001.

[6] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. In T. Henzinger and C. Kirsch, editors, *Proceedings of EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 166–184. Springer-Verlag, 2001. Available at www-cad.eecs.berkeley.edu/ tah/Publications/#hybrid.

[7] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.

[8] E. A. Lee. Embedded software. *Advances in Computers*, 56, 2002.

[9] J. Liu. Continuous time and mixed-signal simulation in Ptolemy II. Memo M98/74, UCB/ERL, EECS UC Berkeley, CA 94720, July 1998.

[10] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee. A hierarchical hybrid system model and its simulation. In *38th IEEE conference on Decision and Control, Phoenix, AZ*, December 1999.

[11] J. Ludvig, J. McCarthy, S. Neuendorffer, and S. R. Sachs. Reprogrammable platforms for high-speed data acquisition. *Journal of Design Automation for Embedded Systems*, 7(4):341–364, Nov. 2002.

[12] M. A. A. Sanvido. *Hardware-in-the-loop Simulation Framework*. PhD thesis, ETH Zurich, Mar. 2002.

[13] M. Tiller. *Introduction to Physical Modeling With Modelica*. Kluwer Academic Publishers, 2001.